**https://dzone.com/articles/ioc-vs-dis**

**Spring IoC Tutorial**

In Spring, the *Inversion of Control (IoC)* principle is implemented using the *Dependency Injection (DI)* design pattern. Let's understand dependency injection with the help of an example. First we will see a java version of the example and later we will add spring functionalities to it. As far as the example go, its pretty simple. The *QuizMater* interface exposes the popQuestion() method. To keep things simple, our *QuizMaster* will generate only one question.

QuizMaster.java

----------------

package com.vaannila;

public interface QuizMaster {

public String popQuestion();

}

The *StrutsQuizMaster* and the *SpringQuizMaster* class implements *QuizMaster* interface and they generate questions related to struts and spring respectively.

StrutsQuizMaster.java

----------------------

package com.vaannila;

public class StrutsQuizMaster implements QuizMaster {

@Override

public String popQuestion() {

return "Are you new to Struts?";

}

}

SpringQuizMaster.java

----------------------

package com.vaannila;

public class SpringQuizMaster implements QuizMaster {

@Override

public String popQuestion() {

return "Are you new to Spring?";

}

}

We have a *QuizMasterService* class that displays the question to the user. The *QuizMasterService*class holds reference to the *QuizMaster*.

QuizMasterService.java

-----------------------

package com.vaannila;

public class QuizMasterService {

private QuizMaster quizMaster = new SpringQuizMaster();

public void askQuestion()

{

System.out.println(quizMaster.popQuestion());

}

}

Finally we create the *QuizProgram* class to conduct quiz.

QuizProgram.java

----------------

package com.vaannila;

public class QuizProgram {

public static void main(String[] args) {

QuizMasterService quizMasterService = new QuizMasterService();

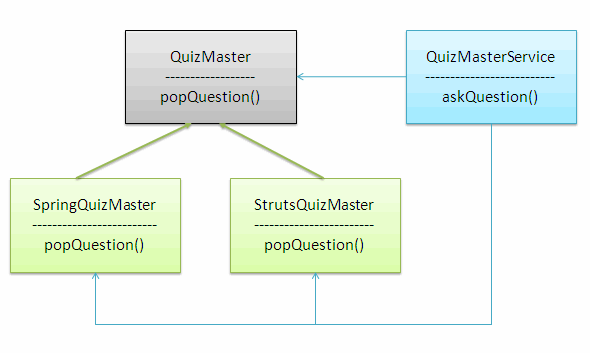
quizMasterService.askQuestion();

}

}

As you can see it is pretty simple, here we create an instance of the *QuizMasterService* class and call the *askQuestion()* method. When you run the program as expected "*Are you new to Spring?*" gets printed in the console.

Let's have a look at the class diagram of this example. The green arrows indicate generalization and the blue arrows indicates association.



As you can see this architecture is tightly coupled. We create an instance of the *QuizMaster* in the *QuizMasterService* class in the following way.

private QuizMaster quizMaster = new SpringQuizMaster();

To make our quiz master Struts genius we need to make modifications to the *QuizMasterService* class like this.

private QuizMaster quizMaster = new StrutsQuizMaster();

So it is tightly coupled. Now lets see how we can avoid this by using the *Dependency Injection* design pattern. The Spring framework provides prowerful container to manage the components. The container is based on the Inversion of Control (IoC) principle and can be implemented by using the Dependency Injection (DI) design pattern. Here the component only needs to choose a way to accept the resources and the container will deliver the resource to the components.

In this example instead of we, directly creating an object of the *QuizMaster* bean in the *QuizMasterService* class, we make use of the container to do this job for us. Instead of hard coding any values we will allow the container to inject the required dependancies.

We can inject the dependancies using the setter or constructor injection. Here we will see how we can do this using the setter injection.

QuizMasterService.java

-----------------------

package com.vaannila;

public class QuizMasterService {

QuizMaster quizMaster;

public void setQuizMaster(QuizMaster quizMaster) {

this.quizMaster = quizMaster;

}

public void askQuestion()

{

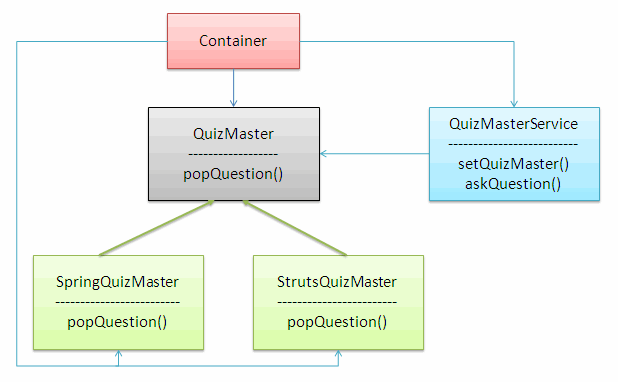
System.out.println(quizMaster.popQuestion());

}

}

The value for the *QuizMaster* will be set using the *setQuizMaster()* method. The QuizMaster object is never instantiated in the *QuizMasterService* class, but still we access it. Usually this will throw a *NullPointerException,* but here the container will instantiate the object for us, so it works fine.

After making all the changes, the class diagram of the example look like this.



The container comes into picture and it helps in injecting the dependancies.

The bean configuration is done in the *beans.xml* file.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation=" http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="springQuizMaster" class="com.vaannila.SpringQuizMaster"></bean>

<bean id="strutsQuizMaster" class="com.vaannila.StrutsQuizMaster"></bean>

<bean id="quizMasterService" class="com.vaannila.QuizMasterService">

<property name="quizMaster">

<ref local="springQuizMaster"/>

</property>

</bean>

</beans>

We define each bean using the *bean* tag. The *id* attribute of the bean tag gives a logical name to the bean and the *class* attribute represents the actual bean class. The *property* tag is used to refer the property of the bean. To inject a bean using the setter injection you need to use the *ref* tag.

Here a reference of *SpringQuizMaster* is injected to the *QuizMaster* bean. When we execute this example, "*Are you new to Spring?*" gets printed in the console.

To make our *QuizMaster* ask questions related to Struts, the only change we need to do is, to change the bean reference in the *ref* tag.

<bean id="quizMasterService" class="com.vaannila.QuizMasterService">

<property name="quizMaster">

<ref local="strutsQuizMaster"/>

</property>

</bean>

In this way the Dependency Injection helps in reducing the coupling between the components.

To execute this example add the following jar files to the classpath.

antlr-runtime-3.0

commons-logging-1.0.4

org.springframework.asm-3.0.0.M3

org.springframework.beans-3.0.0.M3

org.springframework.context-3.0.0.M3

org.springframework.context.support-3.0.0.M3

org.springframework.core-3.0.0.M3

org.springframework.expression-3.0.0.M3

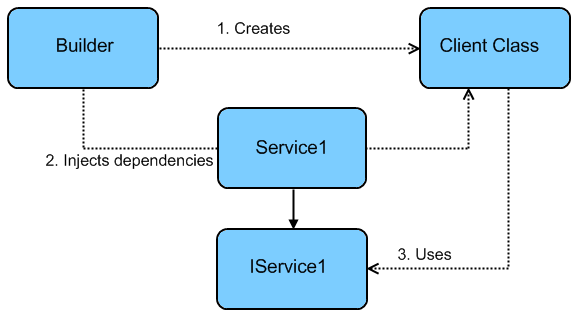
This article explains the concept of Dependency Injection (DI) and how it works in Spring Java application development. You will learn about the advantages, disadvantages, and basics of DI with examples. Look further for more information.

## ****Advantages of Dependency Injection****

* DI allows a client the flexibility of being configurable. Only client's behavior is fixed.
* Testing can be performed using mock objects.
* Loosely couple architecture.
* DI advantages of high cohesion are:
  + Reduced module complexity
  + Increased system maintainability, because logic changes in the domain affect fewer modules.
  + Increased module reusability.
* DI does not require any changes in code behavior it can be applied to legacy code as refactoring.
* DI allows a client to remove all knowledge of a concrete implementation that needs to use. It is more reusable, more testable, more readable code.
* DI makes it possible to eliminate, or at least reduce unnecessary dependencies.
* DI allows concurrent or independent development.
* DI decreases coupling between a class and its dependency.

## ****Disadvantages of Dependency Injection****

* DI creates clients that demand configure details supplied by construction code.
* DI can make code difficult to trace because it separates behavior from construction; this means developers refer to more files to follow how a system performs.
* DI can cause an explosion of types, especially in languages that have explicit interface types like C# and Java.
* DI can encourage dependence on DI framework.
* Tight coupling :
  + A change in only one module usually forces a ripple effect of changes in other modules.



## ****Dependency Injection (DI)****

* Dependency Injection (DI) is a software design pattern that implements inversion of control for resolving dependencies.
* An injection is the passing of a dependency to a dependent object that would use it.
* DI is a process whereby objects define their dependencies. The other objects they work with—only through constructor arguments or arguments to a factory method or property—are set on the object instance after it is constructed or returned from a factory method.
* The container then injects those dependencies, and it creates the bean. This process is named Inversion of Control (IoC) (the bean itself controls the instantiation or location of its dependencies by using direct construction classes or a Service Locator).
* DI refers to the process of supplying an external dependency to a software component.

## Dependency Injection Performed Two Ways

### ****1. Constructor-Based Dependency Injection****

* Constructor-based DI is when the container invokes a constructor with a number of arguments, each of which represents a dependency or other class.
* Calling a static factory method with particular arguments to construct the bean is approximately equivalent, treating arguments to a constructor and to a static factory method. The following example shows a class that can only be dependency-injected with constructor injection. It is a POJO that has no dependencies on container specific interfaces, base classes, or annotations.

public class SimpleStudentList {

// the SimpleStudentList has a dependency on StudentFind

private StudentFind studentFind;

// a constructor that Spring container can 'inject' a StudentFind

public SimpleStudentList(StudentFind studentFind ) {

this.studentFind = studentFind ;

}

// business logic code

}

#### ****Example of Constructor-Based DI****

##### **Book.java**

package com.spring.example;

public class Book {

private int id;

private String bookName;

public Book() {System.out.println("Java");}

public Book(int id) {this.id = id;}

public Book(String bookName) { this.bookName = bookName;}

public Book(int id, String bookName) {

this.id = id;

this.bookName = bookName;

}

void display(){

System.out.println(id+" "+bookName);

}

}

##### **applicationContext.xml**

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="book" class="com.spring.example.Book">

<constructor-arg value="1" type="int"></constructor-arg>

</bean>

</beans>

##### **Main.java**

package com.spring.example;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.\*;

public class Main {

public static void main(String[] args) {

Resource r=new ClassPathResource("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(r);

Book b=(Book)factory.getBean("book");

b.display();

}

}

##### **Output:**

1 null

### ****2. Setter-Based Dependency Injection****

Setter-based DI is the when the container calls setter methods on your beans after it has invoked a no-argument constructor or no-argument static factory method to instantiate that bean.

The following example shows a class that can only have pure setter injection.

public class SimpleStudentList {

// the SimpleStudentList has a dependency on StudentFind

private StudentFind studentFind;

// a setter method that Spring container can 'inject' a StudentFind

public void setStudentFind(StudentFind studentFind ) {

this.studentFind = studentFind ;

}

// business logic

}

#### ****Example of Setter Based DI****

##### **Book.java**

package com.spring.example;

public class Book {

private int id;

private String bookName;

private String author;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getBookName() {

return bookName;

}

public void setBookName(String bookName) {

this.bookName = bookName;

}

public String getAuthor() {

return author;

}

public void setAuthor(String author) {

this.author = author;

}

void display(){

System.out.println(id+" "+bookName+" "+author);

}

}

##### **applicationContext.xml**

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="book" class="com.spring.example.Book">

<property name="id">

<value>1</value>

</property>

<property name="bookName">

<value>The Complete Reference J2EE</value>

</property>

<property name="author">

<value>Herbert Schildt</value>

</property>

</bean>

</beans>

##### **Main.java**

package com.spring.example;

import org.springframework.beans.factory.BeanFactory;

import org.springframework.beans.factory.xml.XmlBeanFactory;

import org.springframework.core.io.\*;

public class Main {

public static void main(String[] args) {

Resource r=new ClassPathResource("applicationContext.xml");

BeanFactory factory=new XmlBeanFactory(r);

Book b=(Book)factory.getBean("book");

b.display();

}

}

##### **Output :**

* The Complete Reference J2EE  Herbert Schildt

Spring Annotation Driven

<https://dzone.com/articles/dependency-injection-an-introd>

### Dependency Injection

Dependency Injection (DI) refers to the process of supplying an external dependency to a software component. DI can help make your code architecturally pure. It aids in design by interface as well as test-driven development by providing a consistent way to inject dependencies. For example, a data access object (DAO) may depend on a database connection. Instead of looking up the database connection with JNDI, you could inject it.   
  
One way to think about a DI container like Spring is to think of JNDI turned inside out. Instead of an object looking up other objects that it needs to get its job done (dependencies), a DI container injects those dependent objects. This is the so-called Hollywood Principle, “Don't call us” (lookup objects), “we’ll call you” (inject objects).   
  
If you have worked with CRC cards you can think of a dependency as a collaborator, i.e., an object that another object needs to perform its role.   
Let's say that you have an automated teller machine (ATM) and it needs the ability to talk to a bank. It uses what it calls a transport object to do this. In this example, a transport object handles the low-level communication to the bank.   
  
This example could be represented by either of the  two interfaces as follows:   
  
**AutomatedTellerMachine interface**

package com.arcmind.springquickstart;

import java.math.BigDecimal;

public interface AutomatedTellerMachine {

void deposit(BigDecimal bd);

void withdraw(BigDecimal bd);

}

**ATMTransport interface**

package com.arcmind.springquickstart;

public interface ATMTransport {

void communicateWithBank(byte [] datapacket);

}

Now the AutomatedTellerMachine needs a transport to perform its intent, namely withdraw money and deposit money. To carry out these tasks, the AutomatedTellerMachine may depend on many objects and collaborates with its dependencies to complete the work.   
  
An implementation of the AutomatedTellerMachine may look like this:

**AutomatedTellerMachine implementation:**

package com.arcmind.springquickstart;

import java.math.BigDecimal;

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine{

private ATMTransport transport;

public void deposit(BigDecimal bd) {

...

transport.communicateWithBank(...);

}

public void withdraw(BigDecimal bd) {

...

transport.communicateWithBank(...);

}

public void setTransport(ATMTransport transport) {

this.transport = transport;

}

}

The AutomatedTellerMachineImpl does not know or care how the transport withdraws and deposits money from the bank. This level of indirection allows us to replace the transport with different implementations such as in the following example:  
  
**Three example transports: SoapAtmTransport, StandardAtmTransport and SimulationAtmTransport**

package com.arcmind.springquickstart;

public class SoapAtmTransport implements ATMTransport {

public void communicateWithBank(byte[] datapacket) {

...

}

}

package com.arcmind.springquickstart;

public class StandardAtmTransport implements ATMTransport {

public void communicateWithBank(byte[] datapacket) {

...

}

}

package com.arcmind.springquickstart;

public class SimulationAtmTransport implements ATMTransport {

public void communicateWithBank(byte[] datapacket) {

...

}

}

Notice the possible implementations of the ATMTransport interface. The AutomatedTellerMachineImpl does not know or care which transport it uses. Also, for testing and developing, instead of talking to a real bank, notice that you can use the SimulationAtmTransport.

The concept of DI transcends Spring. Thus, you can accomplish DI without Spring as follows:

**DI without Spring**

package com.arcmind.springquickstart;

import java.math.BigDecimal;

public class AtmMain {

public void main (String[] args) {

AutomatedTellerMachine atm = new AutomatedTellerMachineImpl();

ATMTransport transport = new SoapAtmTransport();

/\* Inject the transport. \*/

((AutomatedTellerMachineImpl)atm).setTransport(transport);

atm.withdraw(new BigDecimal("10.00"));

atm.deposit(new BigDecimal("100.00"));

}

}

Then injecting a different **transport** is a mere matter of calling a different setter method as follows:

**Injecting a different dependency**

ATMTransport transport = new SimulationAtmTransport();

((AutomatedTellerMachineImpl)atm).setTransport(transport);

To use Spring to inject a dependency you could do the following:

**Using Spring to manage dependencies**

package com.arcmind.springquickstart;

import java.math.BigDecimal;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AtmMain {

public static void main (String[] args) {

ApplicationContext appContext = new ClassPathXmlApplicationContext("classpath:./spring/applicationContext.xml");

AutomatedTellerMachine atm = (AutomatedTellerMachine) appContext.getBean("atm");

atm.withdraw(new BigDecimal("10.00"));

atm.deposit(new BigDecimal("100.00"));

}

}

**/spring/applicationContext.xml file**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="atmTransport" class="com.arcmind.springquickstart.SoapAtmTransport" />

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<property name="transport" ref="atmTransport" />

</bean>

</beans>

Figure 1 illustrates how Spring injects the dependency using property setter method injection.

[img\_assist|nid=5868|title=|desc=|link=none|align=undefined|width=720|height=540]

The application context is the central interface to the Spring DI container. In the application context, you declare two beans, atmTransport and atm, with a bean tag. Then you use the property tag to inject the atmTransport bean into the transport property. This effectively calls the setter method of the AutomatedTellerMachineImpl transport property (setTransport(...)).

The major capabilities that the application context provides include (taken from API docs):

* Bean factory methods for accessing application components
* The ability to load file resources in a generic fashion
* The ability to resolve messages, supporting internationalization

The focus of this article is bean factory methods and DI.

### Using constructor instead of setter

Another option when using Spring is to use constructor arguments instead of setter methods to inject dependencies. This keeps things more pure from an object-oriented design standpoint as an object has to be created with all of its collaborators (a.k.a. dependencies) it needs to fulfill its role.   
  
Using constructors, injection is much like using setter methods as follows:   
  
**Application context for constructor injection**

<bean id="standardTransport" class="com.arcmind.springquickstart.StandardAtmTransport"/>

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<constructor-arg ref="standardTransport" />

</bean>

Notice the use of the constructor-arg tag. This implies that the constructor takes transport as a single argument.   
  
**Adding a constructor to AutomatedTellerMachineImpl**

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine{

private ATMTransport transport;

public AutomatedTellerMachineImpl (ATMTransport transport) {

this.transport = transport;

}

The above example should keep the object purists in your group happy. However, the setter injection style makes test-driven development a bit easier. In practice, the setter method approach is used more often.   
Figure 2 illustrates how the constructor injection occurs.

[img\_assist|nid=5869|title=|desc=|link=none|align=undefined|width=720|height=540]

If you have many constructors in the same class with a variable number of arguments, Spring will try to pick the best fit. However, you can give Spring some hints as follows:   
  
**Application context for constructor injection with a hint for Spring**

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<constructor-arg index="0" ref="standardTransport" />

</bean>

Under some circumstances, you can even specify the type attribute that you want Spring to use to resolve the constructor argument in cases when there are more than one possible constructor match.  Most times, you don't have to specify index or type. Your mileage may vary.

### Spring and Annotation driven DI

Seam, and Guice pioneered the use of DI using annotation instead of XML. Spring also added this support and in typical Spring fashion, it does this in a flexible non-invasive manner.   
  
Let's start off with a simple example. Let's say that you misconfigured the AutomatedTellerMachineImpl and forgot to inject a dependency as follows:   
Opps forgot to inject the transport

<bean id="atmTransport" class="com.arcmind.springquickstart.SoapAtmTransport" />

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

</bean>

You might get an error like this:   
  
**Typical error from misconfiguring a bean**

**Exception in thread "main" java.lang.NullPointerException   
        at com.arcmind.springquickstart.AutomatedTellerMachineImpl.withdraw(AutomatedTellerMachineImpl.java:25)   
        at com.arcmind.springquickstart.AtmMain.main(AtmMain.java:14)**

In a deployed application, this error could be quite cryptic. If you used the @Required annotation, you could ask Spring to scan the beans and look for missing dependencies as follows:  
  
**AutomatedTellerMachineImpl using @Required on the setter method of the transport property**

import org.springframework.beans.factory.annotation.Required;

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine{

private ATMTransport transport;

@Required

public void setTransport(ATMTransport transport) {

this.transport = transport;

}

Now, when you run this after forgetting to configure a transport, you would get this message:

**Caused by: org.springframework.beans.factory.BeanInitializationException: Property 'transport' is required for bean 'atm'**

This is clearer and makes it easier to develop and debug applications. To enable this dependency checking feature, you must use context:component-scan or the context:annotation-config tags. This is discussed in more detail later. Here is the last example using context:annotation-config:

**Application context file using annotation-config tag**

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:annotation-config/>

<bean id="atmTransport" class="com.arcmind.springquickstart.SoapAtmTransport" />

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<property name="transport" ref="atmTransport"/>

</bean>

</beans>

### Using @Autowire to define a default transport

You may want to define a default transport for an AutomatedTellerMachine. You could do this with the @Autowire and @Qualifier annotations as follows:   
  
**Using @Autowire and @Qualifier annotations to do DI**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine{

@Autowired (required=true)

@Qualifier ("standardTransport")

private ATMTransport transport;

 When using Spring annotations for DI, you do not need to have setter methods (or special constructors) any longer. Spring can inject directly into private fields or you have the option of annotating the setter methods instead. The applicationContext for this example looks like this:   
  
**Many transports configured in applicationContext, no injection specified in XML**

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:annotation-config/>

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport" />

<bean id="standardTransport" class="com.arcmind.springquickstart.StandardAtmTransport" />

<bean id="simulationTransport" class="com.arcmind.springquickstart.SimulationAtmTransport" />

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl"/>

</beans>

Notice that no transport for injection is specified in this file. The annotations specify which transport gets injected by default. Figure 3 illustrates injection using this technique.

[img\_assist|nid=5870|title=|desc=|link=none|align=undefined|width=720|height=540]

You could override which bean gets set by using standard Spring injection. In this way, you have a default (standardTransport) that can be overridden. Here is an example of overriding with another transport when you have a setter method for transport.

**Overriding the annotation in the application context file**

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport" />

<bean id="standardTransport" class="com.arcmind.springquickstart.StandardAtmTransport" />

<bean id="simulationTransport" class="com.arcmind.springquickstart.SimulationAtmTransport" />

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<property name="transport" ref="simulationTransport"/>

</bean>

The XML DI injection takes precedence over the annotation. Therefore, the annotation is the "reasonable default", but the application context file has the final word.

### Avoiding hard-wiring beans directly to other beans with @Qualifier and qualifier tag

For an extra level of indirection, you can add a qualifier to a bean in the configuration file and then specify which type of transport is needed in the AutomatedTellerMachineImpl as follows:   
  
**Using @Qualifier for an extra level of indirection**

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine{

@Autowired (required=true)

@Qualifier ("default")

private ATMTransport transport;

**Using qualifier tag in applicationContext.xml**

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport" />

<bean id="standardTransport" class="com.arcmind.springquickstart.StandardAtmTransport">

<qualifier value="default"/>

<!-- NOTE ADDED THIS QUALIFIER that marks this as default -->

</bean>

<bean id="simulationTransport" class="com.arcmind.springquickstart.SimulationAtmTransport" />

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl"/>

With this extra level of indirection, you are not hard-wiring beans directly to other beans, and if you decide that you should use a new default transport object you don't have to rewire every dependent bean.   
Figure 4 illustrates injection using this technique.

### Avoiding XML hell with component-scan tag and @Service, @Component, @Repository annotations

Imagine an application with hundreds of managed objects and the size of the XML configuration file(s). You can manage objects with Spring without putting them in the applicationContext files by marking them with @Service, @Component, or @Repository, and telling Spring where to find the objects. Spring will next scan the classpath looking for these beans and then automatically manage their dependencies.   
To perform this feat, you must configure a context:component-scan tag passing the packages you would like Spring to scan as follows:   
  
**Using component-scan tag in applicationContext.xml**

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="com.arcmind.springquickstart"/>

</beans>

Then you mark your beans with the @Service, @Component, or @Repository as follows:   
  
**AutomatedTellerMachine class using @Service**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Service;

@Service ("atm")

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine{

@Autowired (required=true)

@Qualifier ("default")

private ATMTransport transport;

**Three transports using @Component**

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component("standardTransport")

@Qualifier("default")

public class StandardAtmTransport implements ATMTransport {

public void communicateWithBank(byte[] datapacket) {

...

}

}

@Component("soapTransport")

public class SoapAtmTransport implements ATMTransport {

public void communicateWithBank(byte[] datapacket) {

...

}

}

@Component("simulationTransport")

public class SimulationAtmTransport implements ATMTransport {

public void communicateWithBank(byte[] datapacket) {

...

}

}

Notice that there is a @Qualifier annotation used in the StandardAtmTransport to denote it as the default transport for this application. For new projects, it makes sense to use annotations for objects that don't often change their dependencies. Avoiding XML and using annotation is the new trend in DI; some say it is a best practice. Figure 5 illustrates injection using this technique.

### Configuring objects

In addition to injecting dependencies, Spring allows you to configure objects with primitive and basic types. Let's say that the SoapAtmTransport sometimes has to work in areas where the connection is not so great, so you decide to add a retries property to the SoapAtmTransport as follows:

**SoapAtmTransport with retries**

public class SoapAtmTransport implements ATMTransport {

private int retries=3;

public SoapAtmTransport() {

}

public SoapAtmTransport(int retries) {

this.retries = retries;

}

public void setRetries(int retries) {

this.retries = retries;

}

public void communicateWithBank(byte[] datapacket) {

System.out.printf("SOAP Transport retries %d: %s \n", retries, new String(datapacket));

}

}

Notice that you can pass the retries to the constructor or call the setter method with the number or retries as follows:   
  
**Injecting the number of retries with the setter method**

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport">

<property name="retries" value="5"/>

</bean>

**Injecting the number of retires with a constructor arg**

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport">

<constructor-arg value="6"/>

</bean>

Since this type of configuration is so common, Spring has a shortcut to simplify property value injection as follows:   
  
**Using p namespace in an applicationContext.xml file**

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport" p:retries="7"/>

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<constructor-arg index="0" ref="soapTransport" />

</bean>

</beans>

Notice the use of p:retries="7" is much less verbose than the previous example that used the property tag to set the value. If you are using the Spring IDE plugin for Eclipse, you will get code completion for the p:property-name-syntax. Figure 8 illustrates configuring retries using the shortcut notation added in Spring 2.x.

Spring allows you to configure all primitive types (int, byte, long, etc.), as well as wrapper objects (Integer, Byte, Long, etc.), and many basic types (String, URL, Class, File, etc.).

### Using property place holder configurer

Let's say for each installation of an ATM, the installer may need to configure the number of retries. You probably don't want the installer messing with your XML file for your application context because it is too much like code and too many things could go wrong. Instead, perhaps you could just edit a properties file. The properties file could have properties for each of the things that may vary for a given installation of an AutomatedTellerMachineImpl.

**atm.properties Properties file**

**transport.retries=8**

**applicationContext.xml using property-placeholder tag**

<?xml version="1.0" encoding="UTF-8"?>

<beans

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:property-placeholder location="classpath:atm.properties" />

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport" p:retries="${transport.retries}"/>

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<constructor-arg index="0" ref="soapTransport" />

</bean>

</beans>

Notice the property-placeholder loads the atm.properties file from the classpath. Then you use the transport.retries defined in the atm.properties file as follows: p:retries="${transport.retries}".   
Figure 9 illustrates using the property placeholder configurer.

<context:property-placeholder location="file:./src/main/resources/atm.properties" />

<bean id="soapTransport" class="com.arcmind.springquickstart.SoapAtmTransport" p:retries="${transport.retries}"/>

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl">

<constructor-arg index="0" ref="soapTransport" />

</bean>

### Scopes and lifecycle

Spring supports the concepts of scopes. If you are familiar with JSP and Servlets, you may recall that they have request, session and application scopes. Objects put into request scope stay around for the duration of one request. Objects put into session scope stay around the entire user session (unless destroyed manually) while objects put into application scope stay around as long as the web application is running.

Spring scope support is very similar that of JSP and Servlets. Spring supports the following scopes: prototype, singleton, request, session and more. Plus you can configure you own scope handlers. Outside of a web application, Spring mainly supports two scopes out of the box: prototype and singleton. A singleton-scoped object is the default. It means that the object will stay around as long as the application context does (typically, very similar to application scope in a web application). A prototype scope means that every time that you ask for an object, Spring will create a new one. For example:

**Two atms configured with different scopes**

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl" scope="singleton">

<constructor-arg index="0" ref="soapTransport" />

</bean>

<bean id="atmP" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl" scope="prototype">

<constructor-arg index="0" ref="soapTransport" />

</bean>

If you looked up the atm twice, you would get the same object because it is in singleton scope; however, every time you looked up atmP, you would get a different object because it is in prototype scope. This is demonstrated by the following example:

**Example demonstrating prototype vs. singleton**

AutomatedTellerMachine atm1 = (AutomatedTellerMachine) appContext.getBean("atm");

AutomatedTellerMachine atm2 = (AutomatedTellerMachine) appContext.getBean("atm");

assert atm1 == atm2; //First assert

AutomatedTellerMachine atmP1 = (AutomatedTellerMachine) appContext.getBean("atmP");

AutomatedTellerMachine atmP2 = (AutomatedTellerMachine) appContext.getBean("atmP");

assert atmP1 != atmP2; //Second assert

### Life cycle methods

Often times, you need an object to initialize itself after you have set all of the dependencies. Spring allows you to specify specify an init method as follows:

**Specifying an init method with Spring (applicationContext.xml)**

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl" scope="singleton" init-method="init">

<constructor-arg index="0" ref="soapTransport" />

</bean>

<bean id="atmP" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl" scope="prototype" init-method="init">

<constructor-arg index="0" ref="soapTransport" />

</bean>

Notice the use of the init-method attribute in the bean tag. The name of the method does not have to be init.   
Here is the init method defined in Java. (There are also a few more methods for the transport to add to the flavor of the of the example and a shutdown method which the article will discuss in a moment).

**Init Method and Shutdown method in Java**

public class AutomatedTellerMachineImpl implements AutomatedTellerMachine{

public void init () {

System.out.println("INIT");

transport.connect();

}

public void shutdown () {

System.out.println("SHUTDOWN");

transport.close();

}

The atm bean's init method gets called right after your first load the application context (you can change this by setting the lazy-init attribute to "true"). The prototype atmP bean's init method gets called every time you look it up in the application context.   
  
You can also specify a clean up method using the attribute destroy-method as follows:   
  
**Using destroy-method attribute**

<bean id="atm" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl" scope="singleton"

init-method="init" destroy-method="shutdown">

<constructor-arg index="0" ref="soapTransport" />

</bean>

<bean id="atmP" class="com.arcmind.springquickstart.AutomatedTellerMachineImpl" scope="prototype"

init-method="init" destroy-method="shutdown">

<constructor-arg index="0" ref="soapTransport" />

</bean>

The destroy method would never get called by Spring on atmP because it does not manage the life cycle of prototype beans after creation. The destroy method on atm would only get called if someone gracefully closed the application context which Spring does for some application contexts (this is beyond the scope of this introductory tutorial). Figure 10 illustrates using lifecycle methods.

DI can help make your code architecturally pure. It aids in using a design-by-interface approach as well as test-driven development by providing a consistent way to inject dependencies. You don't need Spring to use DI. You could use DI with plain old Java. However, Spring provides a very nice, powerful DI container.   
There are other DI containers and frameworks out there such as Plexus, Pico container, JBoss microcontainer, and, more recently, Guice. And, other frameworks allow DI like JSF, Seam and more. But, Spring is the de facto industry standard way to do DI.

# Spring Auto scanning components

By [mkyong](http://www.mkyong.com/author/mkyong/" \o "mkyong) | March 24, 2010 | Updated : June 13, 2011 | Viewed : 595,532 times +1,924 pv/w

Normally you declare all the beans or components in XML bean configuration file, so that Spring container can detect and register your beans or components. Actually, Spring is able to auto scan, detect and instantiate your beans from pre-defined project package, no more tedious beans declaration in in XML file.

Following is a simple Spring project, including a customer service and dao layer. Let’s explore the different between declare components manually and auto components scanning in Spring.

## 1. Declares Components Manually

See a normal way to declare a bean in Spring.

Normal bean.

package com.mkyong.customer.dao;

public class CustomerDAO

{

@Override

public String toString() {

return "Hello , This is CustomerDAO";

}

}

DAO layer.

package com.mkyong.customer.services;

import com.mkyong.customer.dao.CustomerDAO;

public class CustomerService

{

CustomerDAO customerDAO;

public void setCustomerDAO(CustomerDAO customerDAO) {

this.customerDAO = customerDAO;

}

@Override

public String toString() {

return "CustomerService [customerDAO=" + customerDAO + "]";

}

}

Bean configuration file (Spring-Customer.xml), a normal bean configuration in Spring.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="customerService" class="com.mkyong.customer.services.CustomerService">

<property name="customerDAO" ref="customerDAO" />

</bean>

<bean id="customerDAO" class="com.mkyong.customer.dao.CustomerDAO" />

</beans>

Run it

package com.mkyong.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App

{

public static void main( String[] args )

{

ApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"Spring-Customer.xml"});

CustomerService cust = (CustomerService)context.getBean("customerService");

System.out.println(cust);

}

}

output

CustomerService [customerDAO=Hello , This is CustomerDAO]

## 2. Auto Components Scanning

Now, enable Spring auto component scanning features.

Annotate with **@Component** to indicate this is class is an auto scan component.

package com.mkyong.customer.dao;

import org.springframework.stereotype.Component;

@Component

public class CustomerDAO

{

@Override

public String toString() {

return "Hello , This is CustomerDAO";

}

}

DAO layer, add **@Component** to indicate this is an auto scan component also.

package com.mkyong.customer.services;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import com.mkyong.customer.dao.CustomerDAO;

@Component

public class CustomerService

{

@Autowired

CustomerDAO customerDAO;

@Override

public String toString() {

return "CustomerService [customerDAO=" + customerDAO + "]";

}

}

Put this “context:component” in bean configuration file, it means, enable auto scanning feature in Spring. The **base-package** is indicate where are your components stored, Spring will scan this folder and find out the bean (annotated with @Component) and register it in Spring container.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="com.mkyong.customer" />

</beans>

Run it

package com.mkyong.common;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App

{

public static void main( String[] args )

{

ApplicationContext context =

new ClassPathXmlApplicationContext(new String[] {"Spring-AutoScan.xml"});

CustomerService cust = (CustomerService)context.getBean("customerService");

System.out.println(cust);

}

}

output

CustomerService [customerDAO=Hello , This is CustomerDAO]

This is how auto components scanning works in Spring.

## Custom auto scan component name

By default, Spring will lower case the first character of the component – from ‘CustomerService’ to ‘customerService’. And you can retrieve this component with name ‘customerService’.

CustomerService cust = (CustomerService)context.getBean("customerService");

To create a custom name for component, you can put custom name like this :

@Service("AAA")

public class CustomerService

...

Now, you can retrieve it with this name ‘AAA’.

CustomerService cust = (CustomerService)context.getBean("AAA");

## Auto Components Scan Annotation Types

In Spring 2.5, there are 4 types of auto components scan annotation types

* @Component – Indicates a auto scan component.
* @Repository – Indicates DAO component in the persistence layer.
* @Service – Indicates a Service component in the business layer.
* @Controller – Indicates a controller component in the presentation layer.

So, which one to use? It’s really doesn’t matter. Let see the source code of @Repository,@Service or @Controller.

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component

public @interface Repository {

String value() default "";

}

You will noticed that all @Repository,@Service or @Controller are annotated with @Component. So, can we use just @Component for all the components for auto scanning? Yes, you can, and Spring will auto scan all your components with @Component annotated.

It’s working fine, but not a good practice, for readability, you should always declare @Repository,@Service or @Controller for a specified layer to make your code more easier to read, as following :

DAO layer

package com.mkyong.customer.dao;

import org.springframework.stereotype.Repository;

@Repository

public class CustomerDAO

{

@Override

public String toString() {

return "Hello , This is CustomerDAO";

}

}

Service layer

package com.mkyong.customer.services;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.mkyong.customer.dao.CustomerDAO;

@Service

public class CustomerService

{

@Autowired

CustomerDAO customerDAO;

@Override

public String toString() {

return "CustomerService [customerDAO=" + customerDAO + "]";

}

}

Spring Auto-Wiring Beans

In Spring framework, you can wire beans automatically with auto-wiring feature. To enable it, just define the “**autowire**” attribute in <bean>.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

In Spring, 5 Auto-wiring modes are supported.

* no – Default, no auto wiring, set it manually via “ref” attribute
* byName – Auto wiring by property name. If the name of a bean is same as the name of other bean property, auto wire it.
* byType – Auto wiring by property data type. If data type of a bean is compatible with the data type of other bean property, auto wire it.
* constructor – byType mode in constructor argument.
* autodetect – If a default constructor is found, use “autowired by constructor”; Otherwise, use “autowire by type”.

## Examples

A Customer and Person object for auto wiring demonstration.

package com.mkyong.common;

public class Customer

{

private Person person;

public Customer(Person person) {

this.person = person;

}

public void setPerson(Person person) {

this.person = person;

}

//...

}

package com.mkyong.common;

public class Person

{

//...

}

## 1. Auto-Wiring ‘no’

This is the default mode, you need to wire your bean via ‘ref’ attribute.

<bean id="customer" class="com.mkyong.common.Customer">

<property name="person" ref="person" />

</bean>

<bean id="person" class="com.mkyong.common.Person" />

## 2. Auto-Wiring ‘byName’

Auto-wire a bean by property name. In this case, since the name of “person” bean is same with the name of the “customer” bean’s property (“person”), so, Spring will auto wired it via setter method – “setPerson(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="person" class="com.mkyong.common.Person" />

## 3. Auto-Wiring ‘byType’

Auto-wire a bean by property data type. In this case, since the data type of “person” bean is same as the data type of the “customer” bean’s property (Person object), so, Spring will auto wired it via setter method – “setPerson(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byType" />

<bean id="person" class="com.mkyong.common.Person" />

## 4. Auto-Wiring ‘constructor’

Auto-wire a bean by property data type in constructor argument. In this case, since the data type of “person” bean is same as the constructor argument data type in “customer” bean’s property (Person object), so, Spring auto wired it via constructor method – “public Customer(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="constructor" />

<bean id="person" class="com.mkyong.common.Person" />

## 5. Auto-Wiring ‘autodetect’

If a default constructor is found, uses “constructor”; Otherwise, uses “byType”. In this case, since there is a default constructor in “Customer” class, so, Spring auto wired it via constructor method – “public Customer(Person person)“.

<bean id="customer" class="com.mkyong.common.Customer" autowire="autodetect" />

<bean id="person" class="com.mkyong.common.Person" />

Spring Autowiring by Name

n Spring, “**Autowiring by Name**” means, if the name of a bean is same as the name of other bean property, auto wire it.

For example, if a “customer” bean exposes an “address” property, Spring will find the “address” bean in current container and wire it automatically. And if no matching found, just do nothing.

You can enable this feature via autowire="byName" like below :

<!-- customer has a property name "address" -->

<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="address" class="com.mkyong.common.Address" >

<property name="fulladdress" value="Block A 888, CA" />

</bean>

See a full example of Spring auto wiring by name.

## 1. Beans

Two beans, customer and address.

package com.mkyong.common;

public class Customer

{

private Address address;

//...

}

package com.mkyong.common;

public class Address

{

private String fulladdress;

//...

}

## 2. Spring Wiring

Normally, you wire the bean explicitly, via ref attribute like this :

<bean id="customer" class="com.mkyong.common.Customer" >

<property name="address" ref="address" />

</bean>

<bean id="address" class="com.mkyong.common.Address" >

<property name="fulladdress" value="Block A 888, CA" />

</bean>

Output

Customer [address=Address [fulladdress=Block A 888, CA]]

With **autowire by name enabled**, you do not need to declares the property tag anymore. As long as the “address” bean is same name as the property of “customer” bean, which is “address”, Spring will wire it automatically.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="address" class="com.mkyong.common.Address" >

<property name="fulladdress" value="Block A 888, CA" />

</bean>

Output

Customer [address=Address [fulladdress=Block A 888, CA]]

See another example, this time, the wiring will failed, caused the bean “addressABC” is not match the property name of bean “customer”.

<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="addressABC" class="com.mkyong.common.Address" >

<property name="fulladdress" value="Block A 888, CA" />

</bean>

Output

Customer [address=null]

Spring Autowiring by Type

In Spring, “**Autowiring by Type**” means, if data type of a bean is compatible with the data type of other bean property, auto wire it.

For example, a “person” bean exposes a property with data type of “ability” class, Spring will find the bean with same data type of class “ability” and wire it automatically. And if no matching found, just do nothing.

You can enable this feature via autowire="byType" like below :

<!-- person has a property type of class "ability" -->

<bean id="person" class="com.mkyong.common.Person" autowire="byType" />

<bean id="invisible" class="com.mkyong.common.Ability" >

<property name="skill" value="Invisible" />

</bean>

See a full example of Spring auto wiring by type.

## 1. Beans

Two beans, person and ability.

package com.mkyong.common;

public class Person

{

private Ability ability;

//...

}

package com.mkyong.common;

public class Ability

{

private String skill;

//...

}

## 2. Spring Wiring

Normally, you wire the bean explicitly :

<bean id="person" class="com.mkyong.common.Person">

<property name="ability" ref="invisible" />

</bean>

<bean id="invisible" class="com.mkyong.common.Ability" >

<property name="skill" value="Invisible" />

</bean>

Output

Person [ability=Ability [skill=Invisible]]

With **autowire by type enabled**, you can leave the ability property unset. Spring will find the same data type and wire it automatcailly.

<bean id="person" class="com.mkyong.common.Person" autowire="byType" />

<bean id="invisible" class="com.mkyong.common.Ability" >

<property name="skill" value="Invisible" />

</bean>

Output

Person [ability=Ability [skill=Invisible]]

Wait, what if you have two beans with same data type of class “ability”?

<bean id="person" class="com.mkyong.common.Person" autowire="byType" />

<bean id="steal" class="com.mkyong.common.Ability" >

<property name="skill" value="Steal" />

</bean>

<bean id="invisible" class="com.mkyong.common.Ability" >

<property name="skill" value="Invisible" />

</bean>

Output

Exception in thread "main" org.springframework.beans.factory.UnsatisfiedDependencyException:

...

No unique bean of type [com.mkyong.common.Ability] is defined:

expected single matching bean but found 2: [steal, invisible]; nested exception is

org.springframework.beans.factory.NoSuchBeanDefinitionException:

No unique bean of type [com.mkyong.common.Ability] is defined:

expected single matching bean but found 2: [steal, invisible]

In this case, you will hits the UnsatisfiedDependencyException error message.

**Note**  
In autowiring by type mode, you just have to make sure only one unique data type of bean is declared.

Spring Autowiring by Constructor

In Spring, “**Autowiring by Constructor**” is actually [autowiring by Type](http://www.mkyong.com/spring/spring-autowiring-by-type/) in constructor argument. It means, if data type of a bean is same as the data type of other bean constructor argument, auto wire it.

See a full example of Spring auto wiring by constructor.

## 1. Beans

Two beans, developer and language.

package com.mkyong.common;

public class Developer {

private Language language;

//autowire by constructor

public Developer(Language language) {

this.language = language;

}

//...

}

package com.mkyong.common;

public class Language {

private String name;

//...

}

## 2. Spring Wiring

Normally, you wire the bean via constructor like this :

<bean id="developer" class="com.mkyong.common.Developer">

<constructor-arg>

<ref bean="language" />

</constructor-arg>

</bean>

<bean id="language" class="com.mkyong.common.Language" >

<property name="name" value="Java" />

</bean>

Output

Developer [language=Language [name=Java]]

With **autowire by constructor enabled**, you can leave the constructor property unset. Spring will find the compatible data type and wire it automatcailly.

<bean id="developer" class="com.mkyong.common.Developer" autowire="constructor" />

<bean id="language" class="com.mkyong.common.Language" >

<property name="name" value="Java" />

</bean>

Output

Developer [language=Language [name=Java]]

# IoC vs. DI

### Some people use the terms Inversion of Control and Dependency Injection interchangeably. Let's see how they're different and how they can work together.

This post aims to explain both ideas in a simple way.

## Inversion of Control

### Basic Concepts

Here is an informal definition of IoC: “IoC is when you have someone else create objects for you.” So instead of writing “**new MyObject”** in your code, the object is created by someone else. This ‘someone else’ is normally referred to as an IoC container.

This simple explanation illustrates some very important ideas:

1. It is called IoC because control of the object is inverted. It is not the programmer, but someone else who controls the object.
2. IoC is relative in the sense that it only applies to some objects of the application. So there may be IoC for some objects, whereas others are under the direct control of the programmer.

Apart from Spring, there are other examples of IoC like **Java Servlets** and **Akka Actors**.

### The Details

Let’s delve a little more into the definition of IoC. IoC is much more than object creation: a Spring Context or a Servlet Container not only create objects, but manage their entire **lifecycle**. That includes creating objects, destroying them, and invoking certain methods of the object at different stages of its lifecycle. These methods are often described as **callbacks**. Notice again the terminology: methods invoked by the container are callbacks, as opposed to the **direct calls** that programmers make on their own code.

All the IoC containers previously mentioned implement some kind of lifecycle: [Spring Bean Lifecycle](https://www.concretepage.com/spring/spring-bean-life-cycle-tutorial), [Servlet Lifecycle](http://tutorials.jenkov.com/java-servlets/servlet-life-cycle.html" \t "_blank), and [Akka Actor Lifecycle](https://alvinalexander.com/scala/understand-methods-akka-actors-scala-lifecycle" \t "_blank).

Another thing to consider is that, although programmers relinquish their control on the objects, they still need to define the templates used by the IoC container to create said objects.

For instance, in Spring, classes are annotated with @Service or @Component (among many others) to indicate that the Spring Container is to manage the instances of those classes (it is also possible to use XML configuration instead of annotations). Spring-managed objects, as you likely know, are called Beans.

In a Servlet application, any class implementing the [Servlet](https://docs.oracle.com/javaee/6/api/javax/servlet/Servlet.html" \t "_blank) interface will be managed by the Servlet Container.

In an Akka application, the IoC container is called [ActorSystem](https://doc.akka.io/api/akka/2.5/akka/actor/ActorSystem.html" \t "_blank) and the managed objects are instances of classes extending the trait [Actor](https://doc.akka.io/api/akka/2.5/akka/actor/Actor.html) and created through configuration objects called [Props](https://doc.akka.io/api/akka/2.5/akka/actor/Props.html).

Here is a quick summary of the ideas discussed so far:

1. IoC containers control and manage the lifecycle of some objects: creation, destruction, and callback invocations.
2. The programmer must identify the classes whose instances are to be managed by the IoC container. There are several ways to do this: with annotations, by extending some specific classes, using external configuration.
3. The programmer can influence, to some extent, the way the objects are managed by the IoC container. Normally, this is achieved by overriding the default behavior of the object callbacks.

| **IOC CONTAINER** | **MANAGED OBJECT NAME** | **MANAGED OBJECTS DEFINITION** |
| --- | --- | --- |
| Spring Container | Bean | Classes defined with annotations/XML configuration |
| Servlet Container | Servlet | Classes implementing interface Servlet |
| Actor System | Actor | Classes extending trait Actor |

So far, we have managed to explain IoC without needing to talk about Dependency Injection.

## Dependency Injection

Dependency Injection has become one of the cornerstones of modern software engineering, as it is fundamental to allow proper testing. To put it simply, having DI is the opposite to having hardcoded dependencies.

//Hardcoded dependency

public class MyClass {

private MyDependency myDependency = new MyDependency();

}

//Injected dependency

public class MyClass {

private MyDependency myDependency;

public MyClass(MyDependency myDependency){

this.myDependency = myDependency;

}

}

A dependency can be injected in several ways, like a parameter in the constructor or through a “set” method.

As important as DI is, there is a downside to its use, namely: management of dependencies is inconvenient. Let’s take a look at an example: MyClass1 depends on MyClass2, that in turns depends upon MyClass3:

public class MyClass3 {

public void doSomething(){}

}

//MyClass2 depends on MyClass3

public class MyClass2 {

private MyClass3 myClass3;

public MyClass2(MyClass3 myClass3){

this.myClass3 = myClass3;

}

public void doSomething(){

myClass3.doSomething();

}

}

//MyClass1 depends on MyClass2

public class MyClass1 {

private MyClass2 myClass2;

public MyClass1(MyClass2 myClass2){

this.myClass2 = myClass2;

}

public void doSomething(){

myClass2.doSomething();

}

}

public class Main {

public static void main(String[] args) {

//All dependencies need to be managed by the developer

MyClass3 myClass3 = new MyClass3();

MyClass2 myClass2 = new MyClass2(myClass3);

MyClass1 myClass1 = new MyClass1(myClass2);

myClass1.doSomething();

}

}

Now, let’s assume that further down the line, MyClass2 needs a new dependency: MyClass4. We need to make changes to account for this new dependency:

public class MyClass2 {

private MyClass3 myClass3;

private MyClass4 myClass4;

public MyClass2(MyClass3 myClass3, MyClass4 myClass4){

this.myClass3 = myClass3;

this.myClass4 = myClass4;

}

public void doSomething(){

myClass3.doSomething();

myClass4.doSomething();

}

}

public class Main {

public static void main(String[] args) {

MyClass4 myClass4 = new MyClass4();

MyClass3 myClass3 = new MyClass3();

MyClass2 myClass2 = new MyClass2(myClass3, myClass4);

MyClass1 myClass1 = new MyClass1(myClass2);

myClass1.doSomething();

}

}

Although the situation described in this example is not too bad, real-life applications can have hundreds of dependencies scattered all across the codebase whose creation and management would need to be centralized like in the above example.

## Inversion of Control and Dependency Injection Playing Together

We just discussed the issue of managing hundreds of dependencies in a real-life application, possibly with very complicated dependency graphs.

So here is where IoC comes to the rescue. With IoC, the dependencies are managed by the container, and the programmer is relieved of that burden.

Using annotations like @Autowired, the container is asked to inject a dependency where it is needed, and the programmers do not need to create/manage those dependencies by themselves.

public class MyClass1 {

@Autowired

private MyClass2 myClass2;

public void doSomething(){

myClass2.doSomething();

}

}

public class MyClass2 {

@Autowired

private MyClass3 myClass3;

@Autowired

private MyClass4 myClass4;

public void doSomething(){

myClass3.doSomething();

myClass4.doSomething();

}

}

## Conclusion

We have presented Inversion of Control and Dependency Injection as separate concepts and justified how in some situations both concepts can be combined to provide superior solutions.

https://dzone.com/articles/a-guide-to-spring-framework-annotations